APPLICATION PROGRAMMING (2) ASSIGNMENT

NGUGI ANITA WAMBUI

SCT221-0784/2022

1. James is working on a calculator application. Write a C# program that performs addition, subtraction, multiplication, and division on two numbers provided by the user.

```
using System;


class Calculator
{
    static void Main(string[] args)
    {
        Console.Write("Enter first number: ");
        double num1 = Convert.ToDouble(Console.ReadLine());

        Console.Write("Enter second number: ");
        double num2 = Convert.ToDouble(Console.ReadLine());

        Console.WriteLine("Addition: " + (num1 + num2));
        Console.WriteLine("Subtraction: " + (num1 - num2));
        Console.WriteLine("Multiplication: " + (num1 * num2));
        Console.WriteLine("Division: " + (num1 / num2));
    }
```

}

a. **James wants to add a feature to calculate the average of a list of integers. Write a method in C# that takes an array of integers and returns the average of the scores. Handle edge cases such as an empty array by returning 0.**

```
public double CalculateAverage(int[] numbers)
{
    if (numbers == null || numbers.Length == 0)
        return 0;

    double sum = 0;
    foreach (int number in numbers)
    {
        sum += number;
    }
    return sum / numbers.Length;
}
```

b. **Mary is developing a system to track object creation. Describe the role of constructors in class instantiation and how they differ from other methods. Illustrate the explanation with a class that includes a default constructor and an overloaded constructor.**

Constructors are special methods in a class that are called when an object of the class is instantiated. They initialize the object and set up its initial state. Constructors differ from other methods because they don't have a return type, and their name is always the same as the class name. They can be overloaded to allow different ways of creating an object.

```
class Example
{
    public string Name { get; set; }
    public int ID { get; set; }

    // Default constructor
    public Example()
    {
        Name = "Unknown";
        ID = 0;
    }

    // Overloaded constructor
```

```
    public Example(string name, int id)
    {
        Name = name;
        ID = id;
    }
}
```

c. Sam is creating an employee management system. Create a class Employee with a constructor that takes an employee's name and ID. Demonstrate how to create an instance of the class and include a secondary constructor that accepts optional parameters like department and salary.

```
class Employee
{
    public string Name { get; set; }
    public int ID { get; set; }
    public string Department { get; set; }
    public double Salary { get; set; }

    // Primary constructor
    public Employee(string name, int id)
    {
        Name = name;
        ID = id;
        Department = "Unassigned";
        Salary = 0;
    }

    // Secondary constructor
    public Employee(string name, int id, string department, double salary)
    {
        Name = name;
        ID = id;
        Department = department;
        Salary = salary;
    }
}

// Example of creating an instance
Employee emp1 = new Employee("John Doe", 123);
```

Employee emp2 = new Employee("Jane Smith", 456, "HR", 60000);

2. Lucy is developing a program to compare string inputs from users. Explain the difference between the == operator and the Equals() method in C#. When should each be used?

Explanation:

- The == operator compares the reference equality for reference types and value equality for value types. When used with strings, it checks if the contents of the strings are the same.
- The Equals() method compares the contents of the strings. It's more flexible and allows specifying a comparison type, such as ignoring case.

Use Cases:

- Use == for a quick comparison when reference or value equality is sufficient.
- Use Equals() when you need more control over the comparison, such as case-insensitive checks.

a. **Predict the output of the following code for a system that compares string values. Explain why each comparison evaluates to either true or false:**

string str1 = "Hello";

string str2 = "Hello";

string str3 = new string(new char[] { 'H', 'e', 'l', 'l', 'o' });

Console.WriteLine(str1 == str2);

Console.WriteLine(str1 == str3);

Console.WriteLine(str1.Equals(str3));

string str1 = "Hello";

string str2 = "Hello";

string str3 = new string(new char[] { 'H', 'e', 'l', 'l', 'o' });

Console.WriteLine(str1 == str2);

```
Console.WriteLine(str1 == str3);

Console.WriteLine(str1.Equals(str3));
```

Predictions:

1. `str1 == str2` will print `True` because both refer to the same string in the interned string pool.
2. `str1 == str3` will print `False` because `str3` is a new instance created on the heap, and the reference is different.
3. `str1.Equals(str3)` will print `True` because `Equals()` checks for the content, which is the same in both `str1` and `str3`.

3. George wants to understand the main components of the .NET Framework for a development project. Explain the role of the Common Language Runtime (CLR) and the Base Class Library (BCL) in the .NET Framework and how they work together to provide a smooth development experience.

Explanation:

- **Common Language Runtime (CLR):** It is the execution engine of the .NET Framework, managing the execution of .NET programs. It handles memory management, security, and exception handling.
- **Base Class Library (BCL):** It is a comprehensive collection of reusable classes, interfaces, and value types that provide a solid foundation for .NET applications. The BCL simplifies the development process by providing a consistent object model for common programming tasks like string manipulation, file I/O, and data access.

Working Together:

- The CLR uses the BCL to execute code efficiently. Developers write code that leverages the BCL, and the CLR handles its execution, ensuring security and performance.

a. In a library management system, a function needs to handle file operations. Write a program that demonstrates the use of System.IO.File to create, read, and write to a file containing a list of books.

```
using System;

using System.IO;
```

```
class LibraryManagement
{
   static void Main(string[] args)
   {
      string filePath = "books.txt";

      // Write to the file
      File.WriteAllText(filePath, "Book1\nBook2\nBook3");

      // Read from the file
      string[] books = File.ReadAllLines(filePath);
      Console.WriteLine("Books in the file:");
      foreach (var book in books)
      {
         Console.WriteLine(book);
      }

      // Append to the file
      File.AppendAllText(filePath, "\nBook4");
      Console.WriteLine("Book4 added to the file.");
   }
}
```

4. Peter needs to track different data types in his application. Explain the difference between value types and reference types in C# and provide examples of each. Discuss scenarios where choosing one type over the other could impact performance or behavior.

Explanation:

- **Value Types:** Stored on the stack, value types contain the actual data. Examples include int, double, and structs.
- **Reference Types:** Stored on the heap, reference types contain a reference (or pointer) to the data. Examples include class, string, and arrays.

Impact on Performance:

- Value types are generally faster to access because they are stored on the stack, which is quicker to access. However, they can lead to stack overflow if they are too large.
- Reference types are more flexible and support features like inheritance and polymorphism but have a slight overhead due to the indirection involved in accessing the data on the heap.

a. Write a C# program that demonstrates the concept of value types and reference types using primitive data types and objects. Include comparisons between int and string arrays and their memory addresses using Object.ReferenceEquals.

```
using System;


class ValueVsReference

{

   static void Main(string[] args)

   {

      // Value type

      int a = 10;

      int b = a;

      b = 20;

      Console.WriteLine($"Value type - a: {a}, b: {b}"); // a: 10, b: 20


      // Reference type

      int[] arr1 = { 1, 2, 3 };
```

```
        int[] arr2 = arr1;

        arr2[0] = 10;

        Console.WriteLine($"Reference type - arr1[0]: {arr1[0]}, arr2[0]: {arr2[0]}"); // arr1[0]: 10, arr2[0]: 10



        // Memory address comparison

        string[] strArr1 = { "Hello" };

        string[] strArr2 = strArr1;

        Console.WriteLine(Object.ReferenceEquals(strArr1, strArr2)); // True

    }

}
```

5. Maria wants to design a class in C# with encapsulation principles. Describe how encapsulation applies to classes and objects in C# and how it can help control access to fields and methods.

Explanation: Encapsulation is the concept of restricting access to certain components of an object and only allowing controlled interaction through well-defined interfaces (like properties or methods). In C#, encapsulation is implemented using access modifiers such as `private`, `protected`, `internal`, and `public`. This helps in protecting the data from unintended interference and misuse.

a. **Maria is creating a system for managing people's data. Create a Person class with private fields for name and age and public properties to encapsulate these fields. Add validation in the properties, such as ensuring age is non negative.**

```
class Person

{

    private string name;

    private int age;
```

```
public string Name
{
    get { return name; }
    set { name = value; }
}

public int Age
{
    get { return age; }
    set
    {
        if (value < 0)
        {
            throw new ArgumentException("Age cannot be negative.");
        }
        age = value;
    }
}

public Person(string name, int age)
{
    Name = name;
    Age = age;
}
```

}

6. John is developing an application that uses arrays and enums. Explain the difference between a single-dimensional array and a jagged array and provide a use case for each.

Explanation:

- **Single-Dimensional Array:** It is a linear array with a fixed length. It is best used when you have a list of items that can be represented in a single row or column (e.g., a list of integers).
- **Jagged Array:** It is an array of arrays where each inner array can have different lengths. It is useful when dealing with irregularly shaped data, like a list of students where each student has a different number of grades.

a. Create a method in C# that takes a two-dimensional array of integers and returns the sum of all its elements. Include support for arrays with irregular shapes or missing values.

```
public int SumOfElements(int[][] jaggedArray)
{
   int sum = 0;
   foreach (int[] innerArray in jaggedArray)
   {
      if (innerArray != null)
      {
         foreach (int element in innerArray)
         {
            sum += element;
         }
      }
   }
   return sum;
}
```

b. Emma is designing a color picker for an art application. Define an enum called Color with values Red, Green, and Blue. Also, define a class Shape with a nested class Circle that uses the enum to determine its color.

```
public enum Color
{
   Red,
   Green,
   Blue
```

```
}

public class Shape
{
   public class Circle
   {
      public Color CircleColor { get; set; }

      public Circle(Color color)
      {
         CircleColor = color;
      }

      public void DisplayColor()
      {
         Console.WriteLine($"The circle's color is {CircleColor}");
      }
   }
}
```

7. Michael is working on a program that needs to handle various exceptions. Describe how exceptions are handled in C# using try, catch, and finally blocks. Discuss best practices and potential pitfalls.

Explanation:

- **try block:** Contains code that might throw an exception.
- **catch block:** Contains code to handle the exception.
- **finally block:** Contains code that will run whether an exception occurs or not, often used for cleanup operations.
- **Best Practices:** Catch specific exceptions, not just `Exception`. Avoid empty `catch` blocks. Use `finally` for releasing resources.
- **Pitfalls:** Overuse of exceptions for control flow, catching general exceptions, and failing to rethrow or log exceptions.

a. **For a list management application, write a C# program that demonstrates handling an exception when trying to access an element outside of the bounds of an array. Introduce nested try-catch blocks for different error types.**

```
using System;
```

```
class ListManagement
{
   static void Main(string[] args)
   {
      int[] numbers = { 1, 2, 3 };
      try
      {
         try
         {
            Console.WriteLine(numbers[3]); // This will throw an exception
         }
         catch (IndexOutOfRangeException ex)
         {
            Console.WriteLine("Index out of bounds. " + ex.Message);
         }
      }
      catch (Exception ex)
      {
         Console.WriteLine("An error occurred. " + ex.Message);
      }
      finally
      {
         Console.WriteLine("Execution completed.");
      }
   }
}
```

8. Chloe wants to determine if a number is even, odd, positive, negative, or zero. Write a C# program that takes an integer input from the user and uses if-else conditions to print the appropriate message.

```
using System;


class NumberCheck

{

   static void Main(string[] args)
```

```
    {
        Console.Write("Enter an integer: ");
        int number = Convert.ToInt32(Console.ReadLine());

        if (number > 0)
            Console.WriteLine("The number is positive.");
        else if (number < 0)
            Console.WriteLine("The number is negative.");
        else
            Console.WriteLine("The number is zero.");

        if (number % 2 == 0)
            Console.WriteLine("The number is even.");
        else
            Console.WriteLine("The number is odd.");
    }
}
```

a. **Explain the differences between while, do-while, and for loops, and provide examples of each. Discuss scenarios where each loop type would be appropriate.**

Explanation:

- **while loop:** Repeats a block of code as long as a condition is true. Use when the number of iterations is unknown.
- **do-while loop:** Similar to while, but it executes the block at least once before checking the condition. Use when the block must run at least once.
- **for loop:** Repeats a block of code a fixed number of times. Ideal for iterating through arrays or when the number of iterations is known.

```
// while loop example
int i = 0;
while (i < 5)
{
   Console.WriteLine(i);
   i++;
}

// do-while loop example
int j = 0;
do
{
   Console.WriteLine(j);
   j++;
} while (j < 5);

// for loop example
for (int k = 0; k < 5; k++)
{
   Console.WriteLine(k);
}
```

b. A sequence generator needs to calculate the factorial of a given number. Write a program using a loop to compute the factorial. Add a twist by calculating the factorial for odd numbers.

```
using System;

class FactorialGenerator
{
   static void Main(string[] args)
   {
      Console.Write("Enter a number: ");
      int number = Convert.ToInt32(Console.ReadLine());

      int factorial = 1;
      for (int i = 1; i <= number; i++)
      {
         if (i % 2 != 0) // Only multiply odd numbers
         {
```

```
            factorial *= i;
        }
    }

    Console.WriteLine($"Factorial of odd numbers up to {number} is {factorial}");
  }
}
```

c. Write a C# program that uses nested loops to print a pattern of asterisks in the shape of a right-angled triangle. Add complexity by adjusting the program to print an inverted triangle.

```
using System;

class TrianglePattern
{
  static void Main(string[] args)
  {
    int n = 5; // Number of rows for the triangle

    // Right-angled triangle
    for (int i = 1; i <= n; i++)
    {
      for (int j = 1; j <= i; j++)
      {
        Console.Write("*");
      }
      Console.WriteLine();
    }

    // Inverted right-angled triangle
    for (int i = n; i >= 1; i--)
    {
      for (int j = 1; j <= i; j++)
      {
        Console.Write("*");
      }
      Console.WriteLine();
    }
  }
```

```
}
```

9. David is designing a program that uses threads for concurrent execution. Explain the role of threads in C#. Discuss the main difference between using the Thread class and the Task class, and provide an example where each would be useful.

Explanation:

- **Threads:** Allow concurrent execution of code, enabling multiple operations to run simultaneously, improving the responsiveness of an application.
- **Thread class:** Used for creating and managing threads manually. It's more low-level, requiring explicit control over thread lifecycle.
- **Task class:** Represents an asynchronous operation. It is more abstract and higher-level, often used with async/await for easier management of concurrency.

Use Cases:

- **Thread class:** Useful for low-level control of thread creation, especially when dealing with legacy code or specific threading requirements.
- **Task class:** Preferred in modern C# development for handling parallelism and concurrency in a more efficient and scalable way.

a. **Write a C# program that demonstrates how to use the Thread class to create and start a new thread. Add complexity by having the main thread synchronize with the new thread and handle thread safety.**

```
using System;
using System.Threading;

class Program
{
   private static object lockObject = new object();
   private static int sharedResource = 0;

   static void Main()
   {
      Thread thread1 = new Thread(IncrementResource);
      Thread thread2 = new Thread(IncrementResource);

      thread1.Start();
      thread2.Start();
```

```
            thread1.Join(); // Wait for thread1 to finish
            thread2.Join(); // Wait for thread2 to finish

            Console.WriteLine("Final value of sharedResource: " + sharedResource);
        }

        static void IncrementResource()
        {
            for (int i = 0; i < 1000; i++)
            {
                lock (lockObject) // Ensure that only one thread can access this block at a time
                {
                    sharedResource++;
                }
            }
        }
    }
```

10. For a news aggregation application, write a C# program that uses the HttpClient class to make a GET request to a public API and display the response. Parse JSON data from the response to display article titles and summaries.

```
using System;

using System.Net.Http;

using System.Threading.Tasks;

using Newtonsoft.Json.Linq; // Make sure to install Newtonsoft.Json package via NuGet


class Program

{

    static async Task Main()

    {

        string url = "https://newsapi.org/v2/top-headlines?country=us&apiKey=YOUR_API_KEY"; // Replace YOUR_API_KEY with an actual API key
```

```
        using (HttpClient client = new HttpClient())
        {
            HttpResponseMessage response = await client.GetAsync(url);
            response.EnsureSuccessStatusCode();
            string responseBody = await response.Content.ReadAsStringAsync();

            JObject json = JObject.Parse(responseBody);
            foreach (var article in json["articles"])
            {
                string title = article["title"].ToString();
                string description = article["description"].ToString();
                Console.WriteLine("Title: " + title);
                Console.WriteLine("Summary: " + description);
                Console.WriteLine();
            }
        }
    }
}
```

a. Write a C# program that opens a file, reads its contents line by line, and then writes each line to a new file. Add a twist by filtering lines based on certain keywords or length.

```
using System;
using System.IO;
```

```
class Program
{
    static void Main()
    {
        string inputFile = "input.txt";  // Replace with your input file path
        string outputFile = "output.txt"; // Replace with your output file path
        string keyword = "important"; // Example keyword for filtering
        int minLength = 10; // Minimum length for a line to be written to the output file

        try
        {
            using (StreamReader reader = new StreamReader(inputFile))
            using (StreamWriter writer = new StreamWriter(outputFile))
            {
                string line;
                while ((line = reader.ReadLine()) != null)
                {
                    if (line.Contains(keyword) && line.Length >= minLength)
                    {
                        writer.WriteLine(line);
                    }
                }
            }
```

```
            Console.WriteLine("Filtering complete. Check the output file.");
        }
        catch (Exception ex)
        {
            Console.WriteLine("An error occurred: " + ex.Message);
        }
    }
}
```

11. Discuss the purpose of packages in C# and how to install and use a NuGet package. Explain how packages can simplify development and ensure code consistency.

Purpose of Packages:

- **Code Reusability**: Packages provide pre-written code that you can use, avoiding the need to write common functionalities from scratch.
- **Simplified Management**: They offer a way to manage and maintain external libraries and dependencies in a standardized manner.
- **Code Consistency**: Using well-maintained packages ensures that you're using tested and reliable code.

Installing and Using a NuGet Package:

Install: Use the NuGet Package Manager in Visual Studio or the Package Manager Console. For example, to install the Newtonsoft.Json package:

```
Install-Package Newtonsoft.Json
```

Use: After installation, you can use the package in your code. For instance, with Newtonsoft.Json:

```
using Newtonsoft.Json;
```

a. **Write a C# program that uses a NuGet package to perform JSON serialization and deserialization. Add a twist by handling complex nested JSON structures.**

```
using System;
using Newtonsoft.Json;
```

```
using System.Collections.Generic;

class Program
{
    public class Article
    {
        public string Title { get; set; }
        public string Summary { get; set; }
        public Author Author { get; set; }
    }

    public class Author
    {
        public string Name { get; set; }
        public string Email { get; set; }
    }

    static void Main()
    {
        var article = new Article
        {
            Title = "Sample Article",
            Summary = "This is a summary of the article.",
            Author = new Author { Name = "John Doe", Email = "john.doe@example.com" }
```

```
        };


        // Serialize
        string json = JsonConvert.SerializeObject(article, Formatting.Indented);
        Console.WriteLine("Serialized JSON:\n" + json);


        // Deserialize
        var deserializedArticle = JsonConvert.DeserializeObject<Article>(json);
        Console.WriteLine("\nDeserialized Object:\n" + "Title: " + deserializedArticle.Title + ",
Author: " + deserializedArticle.Author.Name);
    }
}
```

12. Describe the differences between the List, Queue, and Stack data structures. Provide examples of use cases for each, including scenarios where one data structure may be more appropriate than another.

- **List**: A List in C# is a collection of objects that can be accessed by index. It is a dynamic array, meaning it can grow and shrink in size. Lists are best used when you need a collection that allows for random access and where order matters.

Use Case: A List is suitable for scenarios like maintaining a collection of items in a shopping cart, where items need to be accessed, added, or removed at any position.

- **Queue**: A Queue is a first-in, first-out (FIFO) data structure. Items are added to the end of the queue and removed from the front. It is ideal when you need to process items in the order they were added.

Use Case: A Queue is appropriate for tasks like managing print jobs in a print queue, where jobs should be processed in the order they arrive.

- **Stack**: A Stack is a last-in, first-out (LIFO) data structure. Items are added and removed from the top of the stack. It is useful when you need to reverse order or when processing nested structures.

Use Case: A Stack is ideal for scenarios like undo functionality in a text editor, where the most recent action needs to be reversed first.

a. **Write a program that demonstrates the use of a queue to manage a line of people in a bank. Add a twist by prioritizing certain customers (e.g., VIP) to jump the queue.**

```
using System;
using System.Collections.Generic;

class Program
{
   static void Main()
   {
      Queue<string> bankQueue = new Queue<string>();
      List<string> vipCustomers = new List<string> { "Alice", "Bob" };

      // Regular customers join the queue
      bankQueue.Enqueue("John");
      bankQueue.Enqueue("Jane");

      // VIP customers join the queue but get prioritized
      foreach (string vip in vipCustomers)
      {
         bankQueue.Enqueue(vip);
      }

      // Processing the queue
      while (bankQueue.Count > 0)
      {
         string currentCustomer = bankQueue.Dequeue();
         if (vipCustomers.Contains(currentCustomer))
         {
            Console.WriteLine($"VIP {currentCustomer} is being served.");
         }
         else
         {
            Console.WriteLine($"{currentCustomer} is being served.");
         }
      }
   }
```

```
}
```

13. Discuss inheritance in C#. Describe how to implement it and include access modifiers in the context of inheritance.

Inheritance allows a class (derived class) to inherit fields, methods, and properties from another class (base class). This promotes code reuse and a hierarchical class structure.

- **Public**: Members are accessible everywhere.
- **Protected**: Members are accessible within the base class and its derived classes.
- **Private**: Members are accessible only within the class itself (base class).

a. **Create a base class Animal with a method Speak(). Create a derived class Dog that overrides the Speak() method. Add complexity by including additional derived classes like Cat and Bird and demonstrate polymorphism.**

```
using System;
using System.Collections.Generic;

class Animal
{
   public virtual void Speak()
   {
      Console.WriteLine("Animal makes a sound.");
   }
}

class Dog : Animal
{
   public override void Speak()
   {
      Console.WriteLine("The dog barks.");
   }
}

class Cat : Animal
{
   public override void Speak()
   {
      Console.WriteLine("The cat meows.");
   }
```

```
}

class Bird : Animal
{
   public override void Speak()
   {
      Console.WriteLine("The bird sings.");
   }
}

class Program
{
   static void Main()
   {
      List<Animal> animals = new List<Animal> { new Dog(), new Cat(), new Bird() };

      foreach (Animal animal in animals)
      {
         animal.Speak();
      }
   }
}
```

14. Explain polymorphism in C# and how it can be achieved. Provide examples using base and derived classes.

Polymorphism allows objects of different classes to be treated as objects of a common base class. It is achieved through method overriding (runtime polymorphism) or interfaces. The key advantage is the ability to use a single interface to represent different underlying forms.

- **Method Overriding**: A derived class provides a specific implementation of a method that is already defined in its base class.
- **Interfaces**: Classes can implement multiple interfaces, allowing different implementations of the same method.

a. **Write a program that demonstrates polymorphism using a base class Vehicle and derived classes Car and Bike. Add complexity by including an interface for Drive() and implementing it differently in each derived class.**

```
using System;
using System.Collections.Generic;
```

```
interface IDrive
{
   void Drive();
}

class Vehicle
{
   public virtual void Start()
   {
      Console.WriteLine("Vehicle starts.");
   }
}

class Car : Vehicle, IDrive
{
   public override void Start()
   {
      Console.WriteLine("The car starts.");
   }

   public void Drive()
   {
      Console.WriteLine("The car is driving.");
   }
}

class Bike : Vehicle, IDrive
{
   public override void Start()
   {
      Console.WriteLine("The bike starts.");
   }

   public void Drive()
   {
      Console.WriteLine("The bike is riding.");
   }
```

```
}

class Program
{
   static void Main()
   {
      List<IDrive> vehicles = new List<IDrive> { new Car(), new Bike() };

      foreach (IDrive vehicle in vehicles)
      {
         vehicle.Drive();
      }
   }
}
```

15. Explain abstraction in C# and how it can be implemented using abstract classes and interfaces. Describe scenarios where abstraction can simplify code and enhance maintainability.

Abstraction hides the complex implementation details and shows only the necessary features of an object. It can be implemented using abstract classes and interfaces.

- **Abstract Classes**: Define abstract methods that must be implemented by derived classes and provide common functionality.
- **Interfaces**: Define a contract that implementing classes must follow without providing implementation.

Scenarios: Use abstraction to create a common interface for different types of objects, which allows you to handle them in a uniform way. This simplifies code maintenance and improves scalability.

a. **Create an abstract base class Shape with an abstract method Draw(). Create derived classes Circle and Square that implement the Draw() method. Add complexity by introducing additional properties and methods in derived classes.**

```
using System;

abstract class Shape
{
   public abstract void Draw();
}
```

```
class Circle : Shape
{
   public int Radius { get; set; }

   public override void Draw()
   {
      Console.WriteLine("Drawing a circle with radius " + Radius);
   }
}

class Square : Shape
{
   public int SideLength { get; set; }

   public override void Draw()
   {
      Console.WriteLine("Drawing a square with side length " + SideLength);
   }
}

class Program
{
   static void Main()
   {
      Shape circle = new Circle { Radius = 5 };
      Shape square = new Square { SideLength = 4 };

      circle.Draw();
      square.Draw();
   }
}
```

16. Predict the output of the following code:

int[] array = {1, 2, 3, 4, 5};

for (int i = 0; i < array.Length; i++)

{ Console.WriteLine(array[i]);

}

OUTPUT:

1

2

3

4

5

a Predict the output of the following code:

string str1 = "Hello";

string str2 = "hello";

Console.WriteLine(str1.Equals(str2, StringComparison.OrdinalIgnoreCase));

OUTPUT:

True

b Predict the output of the following code:

object obj1 = new object(); object obj2 = new object();

Console.WriteLine(obj1 == obj2);

OUTPUT:

False

c Predict the output of the following code:

int a = 5; int b = 10;

Console.WriteLine(a += b);

OUTPUT:

15

17. Given a list of integers, write a C# method that returns the largest and smallest integers in the list. Add a twist by handling an empty list.

```
using System;
using System.Collections.Generic;
using System.Linq;


class Program
{
   static void Main()
   {
      List<int> numbers = new List<int> { 10, 2, 33, 4, 5 };
      (int? min, int? max) = FindMinMax(numbers);


      Console.WriteLine("Min: " + min ?? "List is empty");
      Console.WriteLine("Max: " + max ?? "List is empty");
   }


   static (int? Min, int? Max) FindMinMax(List<int> list)
   {
      if (list.Count == 0)
      {
         return (null, null);
      }
```

```
    int min = list.Min();
    int max = list.Max();
    return (min, max);
  }
}
```

a. **Write a C# program that reads integers from the console until a negative number is entered. Calculate and display the sum of all entered integers. Add a twist by ignoring duplicate integers in the sum.**

```
using System;
using System.Collections.Generic;

class Program
{
  static void Main()
  {
    HashSet<int> uniqueNumbers = new HashSet<int>();
    int number;

    Console.WriteLine("Enter integers (negative number to stop):");
    while ((number = int.Parse(Console.ReadLine())) >= 0)
    {
      uniqueNumbers.Add(number);
    }

    int sum = 0;
    foreach (int num in uniqueNumbers)
    {
      sum += num;
    }

    Console.WriteLine("Sum of unique integers: " + sum);
  }
}
```

b. Write a C# program that demonstrates the use of an enum for the days of the week. Add a twist by performing operations based on the enum value (e.g., identifying weekend days).

```
using System;

enum DayOfWeek
{
   Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
}

class Program
{
   static void Main()
   {
      DayOfWeek today = DayOfWeek.Saturday;

      switch (today)
      {
         case DayOfWeek.Saturday:
         case DayOfWeek.Sunday:
            Console.WriteLine("It's the weekend!");
            break;
         default:
            Console.WriteLine("It's a weekday.");
            break;
      }
   }
}
```

c. Write a C# program that takes a string input from the user and prints the string in reverse order. d. Write a C# program that demonstrates how to use the Dictionary class to store and retrieve student grades. Add complexity by handling a variety of data types as keys and values.

```
using System;

class Program
{
   static void Main()
```

```
    {
        Console.WriteLine("Enter a string:");
        string input = Console.ReadLine();
        string reversed = new string(input.Reverse().ToArray());
        Console.WriteLine("Reversed string: " + reversed);
    }
}
```

18. Explain the purpose and benefits of using interfaces in C#. Discuss how interfaces can promote loose coupling and code reusability.

Interfaces define a contract that classes must follow, specifying methods and properties without implementing them. This allows different classes to use the same interface, which promotes loose coupling and code reusability. Interfaces provide a way to ensure that different classes share a common functionality while allowing each class to implement the functionality in its own way.

- **Loose Coupling**: Interfaces help in reducing dependencies between components, making it easier to change or replace components without affecting others.
- **Code Reusability**: Interfaces allow for implementing shared functionalities across different classes, which can be reused wherever needed.

a. **Create an interface IDrive with a method Drive(). Implement this interface in classes Car and Bike. Demonstrate polymorphism by using a list of IDrive objects and calling the Drive() method on each object.**

```
using System;
using System.Collections.Generic;

interface IDrive
{
    void Drive();
}

class Car : IDrive
{
    public void Drive()
    {
        Console.WriteLine("The car is driving.");
    }
}
```

```
class Bike : IDrive
{
   public void Drive()
   {
      Console.WriteLine("The bike is riding.");
   }
}

class Program
{
   static void Main()
   {
      List<IDrive> vehicles = new List<IDrive> { new Car(), new Bike() };

      foreach (IDrive vehicle in vehicles)
      {
         vehicle.Drive();
      }
   }
}
```

b. Explain the role of abstract classes in C# and how they differ from interfaces. Describe scenarios where abstract classes may be more appropriate than interfaces.

Abstract Classes: Abstract classes can provide both complete and incomplete methods (abstract methods). They are used when you want to define common behavior in a base class that derived classes can inherit and extend. Abstract classes can also have fields and constructors.

Interfaces: Interfaces can only define method signatures and properties without implementation. They are used when you want to ensure that different classes implement a set of methods or properties, regardless of their base class.

Scenario: Use abstract classes when you have a common base class that should provide some shared functionality or state to derived classes. Use interfaces when you need to ensure certain behaviors across unrelated classes or when you need to support multiple inheritance.

c. Create an abstract class Animal with an abstract method MakeSound(). Create derived classes Dog and Cat that implement the MakeSound() method. Demonstrate

polymorphism by creating a list of Animal objects and calling MakeSound() on each object.

```
using System;
using System.Collections.Generic;

abstract class Animal
{
   public abstract void MakeSound();
}

class Dog : Animal
{
   public override void MakeSound()
   {
      Console.WriteLine("The dog barks.");
   }
}

class Cat : Animal
{
   public override void MakeSound()
   {
      Console.WriteLine("The cat meows.");
   }
}

class Program
{
   static void Main()
   {
      List<Animal> animals = new List<Animal> { new Dog(), new Cat() };

      foreach (Animal animal in animals)
      {
         animal.MakeSound();
      }
   }
}
```

19. Describe how a project with a top-down approach can benefit from planning the structure and modules of a large-scale application before implementing the lower- level functions. Provide an example project where top-down might be the best approach.

In a top-down approach, you start by designing the high-level structure and architecture of the application before diving into the details of individual components. This approach ensures that the overall system architecture is sound and that the project has a clear roadmap.

Benefits:

- **Clear Architecture**: Helps in understanding the big picture and how different components interact.
- **Better Planning**: Allows for identifying potential issues early in the design phase.
- **Consistency**: Ensures that different parts of the application follow the same design principles and interact smoothly.

Example: Developing an enterprise resource planning (ERP) system where the overall architecture must be defined first to integrate various modules like inventory, finance, and HR. A top-down approach helps ensure that all modules align with the core architecture and work together seamlessly.

a. **In a bottom-up approach, describe how starting with the implementation of small, independent functions and gradually** combining **them into larger units can lead to a more flexible and testable application. Provide an example project where bottom-up might be the best approach.**

In a bottom-up approach, you start by developing and testing individual components or functions before integrating them into larger systems. This approach promotes flexibility and easier testing, as each component is independently tested and verified.

Benefits:

- **Modularity**: Components are developed and tested individually, making it easier to isolate and fix issues.
- **Flexibility**: Changes to one component are less likely to affect others, allowing for easier modifications and updates.
- **Testability**: Smaller components can be thoroughly tested before integration, reducing the risk of bugs in the final system.

Example: Developing a library of reusable utilities and functions, such as a logging framework or data access library. A bottom-up approach allows for creating and testing these utilities independently before integrating them into larger applications.